Building a Trusted Path for Applications Using COTS Components

Hanno Langweg
Norwegian Information Security Laboratory – NISlab[1]
Department of Computer Science and Media Technology, Gjøvik University College
P.O. Box 191, 2802 Gjøvik
Norway

hanno.langweg@hig.no

ABSTRACT

Client computers are often a weak link in a technical network infrastructure. Increasing the security of client systems and applications against malicious software attacks increases the security of the network as a whole.

Our work solves integrity and authenticity of input, confidentiality, integrity and authenticity of output.

We present components to integrate a trusted path into an application to directly communicate with a user at a personal computer. This allows security sensitive parts of applications to continue operating while being attacked with malicious software in an event-driven system. Our approach uses widely employed COTS software – DirectX – and can be varied in design and implementation, hence making it more difficult to defeat with generic attack tools.

1.0 INTRODUCTION

Client computers are often a weak link in a technical network infrastructure. Confidentiality and integrity of connections between nodes in a network can be secured employing strong cryptography. However, this does not help against attacks by malicious software. Trojan horse programs, i.e., programs with additional hidden, often malicious, functions, are more and more popular forms of attack. These assail the endpoints of secured transactions. A vulnerable interface is the interaction of an application with the physically present user.

Examples for a direct user interaction are the creation of electronic signatures, online voting, financial transaction processing, communication software etc. The concept for this is not new and is called a trusted path. A trusted path exists between the physically present user and the operating system, e.g., when invoking the logon process.

In the Clark/Wilson access control model (cf. Clark et al. 1987), integrity is protected by transformation procedures that can not be bypassed. Applications manipulating data in existing systems could be seen as these procedures. If they are not able to tell apart another program from a user, untrusted processes could manipulate data that is assumed to be protected by a special access path.

Current event-driven systems, especially the Microsoft Windows operating system, do provide little to distinguish between users and other processes. Input can be simulated, output can not be authenticated and can be captured by other processes.

[1] This work was partly completed while the author was working at Department of Computer Science III, University of Bonn, Germany

Paper presented at the RTO IST Symposium on "Adaptive Defence in Unclassified Networks", held in Toulouse, France, 19 - 20 April 2004, and published in RTO-MP-IST-041.

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **01 NOV 2004**	2. REPORT TYPE **N/A**	3. DATES COVERED **-**

4. TITLE AND SUBTITLE **Building a Trusted Path for Applications Using COTS Components**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Norwegian Information Security Laboratory NISlab1 Department of Computer Science and Media Technology, Gjøvik University College P.O. Box 191, 2802 Gjøvik Norway**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
See also ADM001845, Adaptive Defence in Unclassified Networks (La defense adaptative pour les reseaux non classifies)., The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**UU**	**28**	

In this paper we deal with integrity and authenticity of input, confidentiality, integrity and authenticity of output. We use existing COTS (commercial off the shelf) software, originally deployed as an interface for game developers, to assemble a trusted path. It is offered as a set of components to application developers.

This paper is organized as follows. We discuss previous and related work and give an overview of the standard Microsoft Windows input and output model. We then present components obtaining input and output by different means. This is followed by a discussion on retrofitting existing applications with a trusted path. An examination of the security of our approach concludes the presentation.

It may be important to note that we focus on architectural vulnerabilities of platforms and applications. We do not cover vulnerabilities that stem from flaws in an implementation.

2.0 PREVIOUS AND RELATED WORK

User interface security has always been an issue. In the Orange Book (1985) a Trusted path was required to establish a secure communication between the user and the operating system. It is there defined as follows: "Trusted Path – A mechanism by which a person at a terminal can communicate directly with the Trusted Computing Base. This mechanism can only be activated by the person or the Trusted Computing Base and cannot be imitated by untrusted software."

Wiseman et al. (1988) propose a user interface for the SMITE system to prevent Trojan horses from tampering with an application's output. Rooker (1993) questions a focus on the operating system. In his view applications should play a more active role in enforcing security. Operating systems should provide object-oriented support for trusted user interfaces and security embedded in applications.

In the Microsoft Windows operating system, applications typically receive information about user actions by way of messages. Since these messages can be sent by malicious applications as well, this turns out to be a convenient vector of attack.[2] It is a vulnerability by design. In Microsoft's view, all applications assigned to the same desktop are treated equally. If a program needs an undisturbed interface, it should be assigned a separate desktop. This approach is pursued by Balfanz (2001). However, managing separate desktops can be cumbersome for developers. So most of today's software that interacts with a user sitting at the machine runs in a single desktop shared by benign and malign programs.

This problem is encountered by local security applications such as virus scanners, personal fire walls etc. Schmid et al. (2002) point out their dilemma when notifying the user about a security event. The user is notified about the presence of a possibly malicious application that could hide that notification instantaneously. Xenitellis (2002a, 2002b) discusses Windows messages in event-driven systems in general and laments a lack of authentication. He proposes a rigorous filtering of messages that could be harmful to an application or separating applications from each other, thereby reducing co-operation among them. The straight-forward alternative, outlined by Spalka et al. (2002), would be to add an authenticated origin to messages. It would require changes in the decade-old messaging system and is hence unlikely to be adopted by the manufacturer of the operating system. In the X-Windows system, a radical approach is pursued, allowing to disable conveyance of all messages placed by the *SendEvents* function (cf. Bråthen 1998). There may be occasions, like computer-based training, in which remote control of another application or parts of it is desired. Only a fraction of all applications expose an interface by which they can be explicitly automated. Consequently, simulating user input is a quick and convenient way for small helper applications.

Paget (2002) and Howard (2002) remind us that messages can be sent between processes running in different security contexts. Says Howard: 'In the Windows user interface, the desktop is the security

[2] Cult of the Dead Cow (2003). *Back Orifice 2000.* http://bo2k.sourceforge.net

boundary, and any application running on the interactive desktop can interact with any window on the interactive desktop, even if that window is invisible. This is true regardless of the security context of the application that creates the window and the security context of the application.'

Carlisle et al. (2001) investigate some improvements for dialog-based security. Application output should be defended against hiding. Actions should be delayed so that a user could interfere when a program is controlled by simulated input. Scripting and automatic collection of information about the user interface should be restricted. Langweg (2002) uses the DirectX interface to distinguish between key strokes and simulated Windows messages. Output is orchestrated by DirectX instead of the co-operative Windows GDI. Ye et al. (2002) advocate a modified web browser to convey meta-information to the user about which browser windows can be trusted.

3.0 ARCHITECTURAL OVERVIEW

3.1 Windows Input Model

Microsoft Windows uses an internal messaging model to control Windows applications. Messages are generated whenever an event occurs. For example, when a user presses a key on the keyboard and releases it or moves the mouse, a message is generated by the operating system. The message is then placed in the message queue for the appropriate thread. An application checks its message queue to retrieve messages.[3]

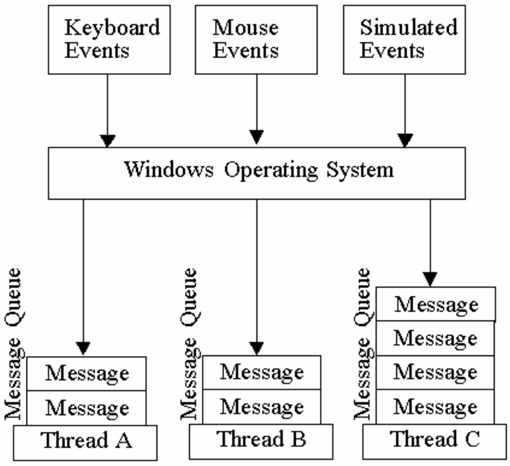


Figure 1: Input processing on a Windows desktop

The system passes all input for an application to the various windows in the application. Each window has a function, called a window procedure, that the system calls whenever it has input for the window. The window procedure processes the input and returns control to the system. All aspects of a window's appearance and behaviour depend on the window procedure's response to these messages.

In the model, it is not possible to distinguish between messages placed in the queue by the operating system and messages placed by another application. To make it even worse, ordinary programs can synthesize input by help of the SendInput API function (keybd_event, mouse_event prior to NT4 SP3). This synthesized input is processed by the operating system into messages for an application. This was originally intended to assist users in operating an application by different input facilities other than the

[3] Microsoft (1998). *Microsoft Windows Architecture for Developers Training Kit.*

standard keyboard and mouse, e.g., assistive technology for users with disabilities. It is also a convenient tool for malicious programs.

3.2 DirectX

Microsoft DirectX is a group of technologies designed by Microsoft to make Microsoft Windows-based computers an ideal platform for running and displaying applications such as games. Built directly into Windows operating systems, DirectX is an integral part of Windows 98, Windows Me, and Windows 2000/XP.

DirectX gives software developers a consistent set of APIs that provides them with improved access to hardware. These APIs control what are called "low-level functions," including graphics memory management and rendering, and support for input devices such as joysticks, keyboards, and mice. The low-level functions are grouped into components that make up DirectX: Microsoft Direct3D, Microsoft DirectDraw, Microsoft DirectInput, to name just a few.[4] In this paper we are concerned with DirectInput and DirectDraw.

DirectInput retrieves information before it is distilled by the operating system to Windows messages. Hence, input synthesized by placing a forged message in a program's message queue is ignored.

DirectDraw allows to access the display hardware in exclusive mode, keeping other programs from distorting the information presented to the user.

In the sketch it is shown how an application actually transfers its output to the screen. Without DirectX it uses the GDI (Graphical Device Interface) and the DDI (Display Driver Interface). With DirectX, namely its DirectDraw part, the DDI is bypassed in favour of the HAL (Hardware Abstraction Layer). If there is no direct hardware support, the HEL (Hardware Emulation Layer) is used instead.[5]

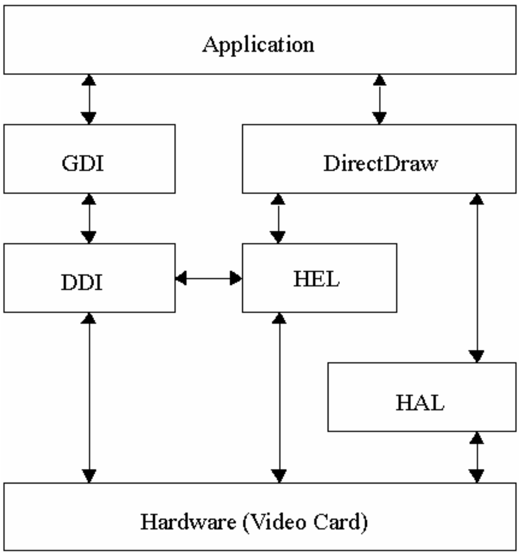


Figure 2: Output processing on a Windows desktop

[4] Microsoft (2000). 'Microsoft DirectX Overview'. *Microsoft Developer Network Library.*

[5] Microsoft (2001). 'DirectDraw Architecture, System Integration'. *Microsoft Developer Network Library.*

4.0 TRUSTED PATH FOR APPLICATIONS

A couple of applications need to communicate directly with the user. This includes situations where it is eminent to get input from the user being physically present at the machine, or to ensure that the user sees data undisturbed by other applications.

Examples are creation of electronic signatures, online voting, financial transactions, communication software etc. The concept for this is the trusted path. A trusted path exists between the user and the operating system, e.g., when invoking the logon process.

For an implementation of a trusted path for security-aware software we build on Langweg's (2002) work. In this approach, DirectX is used to determine whether input was initiated by a device or by a simulated message. Integrity and authenticity of user input is achieved that way.

4.1 Input Components

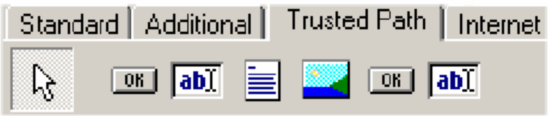


Figure 3: Components palette for input and output

We provide components for application developers. They are shown in the component palette above and comprise two kinds of buttons, two kinds of edit boxes, a memo box, and components for output that will be discussed in a later section. The components are implemented using Borland Delphi.

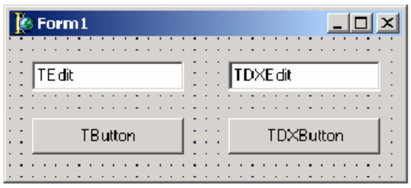


Figure 4: Form showing both standard and trusted path components

These components can be used like the standard Windows controls. In the case of input, we have three different components, an edit control, a memo box, and a button. They offer almost the same functionality as the original controls. Since they inherit from the original TEdit, TMemo and TButton classes, they can replace these components in existing projects.

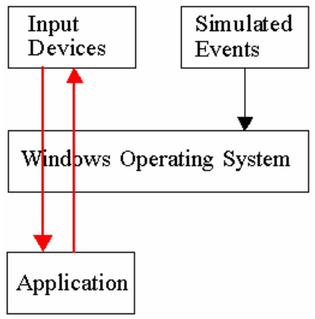


Figure 5: Input processing using DirectX

The new edit control, TDXEdit, does not rely on Windows messages being accurate. It confirms if input can be verified via the DirectInput interface, originally added for game development.

We have two different implementations. In the first, we observe the message queue for input messages. As soon as an input message arrives, we check the DirectInput keyboard state. If a key press or release can be confirmed, we process the input message; otherwise it is discarded. In situations where there is high load on the system, the user may have to type less fast than usual. This did not present a practical problem in tests with most users. However, sometimes genuine input was discarded when it should not have been.

In our second implementation, we process input in a separate thread. This thread observes the DirectInput device either by polling or by a call back function. When input is detected at the DirectInput interface, a message is composed and posted to the message queue. This message is specified as, e.g., WM_DX_KEYDOWN. Its parameters contain an index and a pointer to the input data. The parameters are encrypted using AES to defend against other processes composing a similar message. In the message processing loop, the WM_DX_KEYDOWN parameters are decrypted, and the keyboard input data retrieved from memory. Then a conventional WM_KEYDOWN message is constructed internally and processed by the default method. All other WM_KEYDOWN messages that arrive directly and not as a WM_DX_KEYDOWN are discarded because they could have been manipulated. This holds true for WM_KEYDOWN, WM_KEYUP, WM_CHAR, WM_PASTE. WM_SETTEXT and WM_GETTEXT are also processed only when they originate from inside the process.

We also have a TEdit variant that works with the API function GetAsyncKeyState instead of DirectInput. Its implementation is almost identical to the version presented above.

Our TDXMemo component works similar to the TDXEdit but offers multiple lines for user input.

The TDXButton component offers a button that can not be clicked by a simulated message. The implementation is simpler since only WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK, WM_MOUSEMOVE messages have to be observed. We also have two variants as described above, the first validating messages when they arrive, the second generating messages based on DirectInput and discarding all others.

AES was chosen with respect to high speed in software. The overhead of encrypting and decrypting the input messages is remarkably low. On our 850 MHz PIII time spent on message processing increases by less than two percent in the case of message validation. In the second implementation we observe that message processing takes almost twice as long as in the unencrypted case.

4.2 SendInput

The problem of simulated Windows messages solved, input can in principle still be fabricated. The operating system allows processes running in the same desktop to simulate input by the Win32 API function SendInput.

To prohibit other (possibly malicious) programs to call SendInput and synthesize the user's key strokes we can inject control code into running processes. This code resides in a dynamic link library (DLL) which is activated when USER32.DLL is loaded. Since SendInput is a USER32 function, loading of our DLL is assured. It is necessary to add our DLL to the key `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\APPInit_DLLs` in the Windows registry.[6] This requires administrative privileges during installation.

The DLL modifies the Import Address Table of the supervised program and redirects all calls to SendInput to its own version of that function. Calls can then be blocked or forwarded to the original SendInput function when blocking is not required. The same effect could be achieved by employing a COTS sandbox software for programs running on the computer that monitors and restricts the use of certain API calls including SendInput.

It may be possible to distinguish users and untrustworthy programs by observing their input behaviour, e.g., programs simulating input much faster than an ordinary user could type. This rather falls in the field of biometrics (cf. e.g. Bergadano et al. 2002).

We have found another way to tackle the SendInput problem. We call the first method 'Fast Hook Renewal'. Our input components install a system-wide low-level keyboard hook to catch all input before it is processed by applications. Here we check the LLKHF_INJECTED flag for injected input. If it is set, we discard that input so that it does not reach our secure components. Since other (and malicious) applications can employ this method as well to promote their agenda, we renew our hook after a short period, i.e., some milliseconds, to get to the beginning of the hook chain. This method works quite reliably, albeit not 100% of the time.

Our second approach is modifying the desktop's ACL (access control list). It is possible to enable system-wide hooks for certain accounts only. Hence, ordinary applications could be denied using system-wide hooks while our protected application could use them. A desktop's ACL has to be modified before a (possibly malicious) process is started. So, the canonical point to do this is the creation of the desktop which is the responsibility of the GINA (Graphical Identification and Authentication) library. This cannot be circumvented by an application. However, it requires replacing either the standard GINA of the operating system or that of a third party vendor, e.g., with a smart card or biometric authentication. Currently, creating a chain of GINA libraries does not add to overall system stability.

Eliminating faked messages 100% of the time and discarding injected SendInput data most of the time is still an advantage. It can be done easily be changing some components and leaving the rest of a program untouched. Messages allow an attacker to exactly specify which parts of the user interface should receive input. Achieving the same with SendInput is harder to do. In lack of an appropriate metric, it is not known how much harder exactly, though. Attacks by SendInput can also be countered by introducing delays in the input process (cf. Carlisle et al. 2001). Attacks are not prevented then, but made detectable by the user (or by the application if it suspects the user typing so fast that input can only come from a malicious program). Changing the user interface slightly from time to time also makes it harder for an attacker to use SendInput as a tool.

[6] Microsoft (2000) 'Working with the APPInit_DLLs Registry Value'. *Microsoft Knowledge Base Q197571.*

4.3 Output Components

We provide components for application developers. They are shown in the component palette in fig. 3 and comprise a couple of input components and three output components, one of which is shown as a picture. The components are implemented using Borland Delphi.

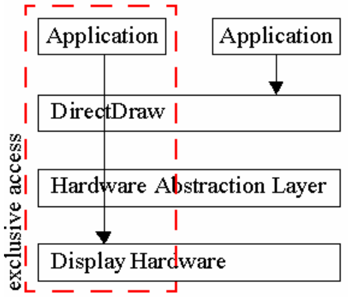
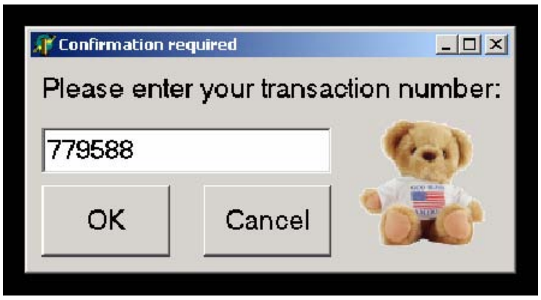


Figure 6: Output processing using DirectX

We have two goals as regards output components. On the one hand, integrity and authenticity of the output has to be ensured. On the other, we want to combine a secure display with the user being able to respond along a trusted path.

Hence, we provide three components. Two display the content of a window on a secure surface. The third offers a standardized limited functionality for user input in combination with a secure surface to draw on.

The first output component, DXFormShow, is used in conjunction with a form's show method. We activate DirectDraw, acquire the screen exclusively, clear the screen, paint it black and draw the form's content on the centre of the screen. As shown in previous work, this ensures a display that can not be manipulated by other processes. In addition, it provides confidentiality of the output. An application developer needs to drop the component on a form and replace calls to a form's show method. The rest of the application's code, including the code used to draw the form, can be left unchanged. However, this only ensures that a form is drawn on a secure surface. If the form contains sophisticated input controls these can not be used by the user. So, we recommend this component only for displaying, e.g., details of a financial transaction or data to be signed when no modification is essential.


Confirmation required

Please enter your transaction number:

779588

Figure 7: Component DXEnhancedMessageBox with simple user interface and application hologram

Our second component used for output, DXMessageBox, is a replacement for the standard Win32 API function MessageBox. A developer provides a caption and text to display in the message box, and the buttons that are shown to the user. We then switch to a secure surface as described above and show the message box. The buttons employed are TDXButtons that we use for trusted input. The user then has a choice of which button to click to give the application a response to the output.

In a variant of the message box we offer an enhanced message box, DXEnhancedMessageBox. In addition to displaying text and offering buttons, the box allows one TDXEdit control to be used. The user may there enter commands securely in text mode.

4.4 Window Personalization

Authenticity of the output is done by window personalization. This concept is described by Tygar et al. (1996). At installation time the user adopts a picture that is displayed each time our application invokes the trusted display mode. Other applications do not have access to the picture and the user can thus determine whether or not to accept the output.

The NTFS file system of Windows NT/2000/XP only allows to specify access rights differentiating users. We put a service on top that identifies processes that request the secret picture. This service runs under a separate account. Access to files containing a picture identifying an application is restricted to the account of the service. Hence, confidentiality and integrity of this picture is protected. We call it an 'application hologram' (e.g., the teddy bear in the example in fig. 7).

The protocol for communicating with our service contains four steps.

- An application opens a named pipe to the service requesting access to the secret graphic. The request contains a handle of a window of the requesting process.
- The service checks which executable module belongs to the window identified by the handle. If the executable is not in the service's white list of benign processes, the protocol stops and an entry is added to the security log. The check could also involve whether the executable file has been tampered with, but this lies outside the scope of our tool. If the service decides that the window belongs to a benign process, it sends a random 32 bit value (a nonce) wrapped in a special message to the window.
- The receiving window checks whether it requested access to the secret graphic. If it decides to proceed, it sends the received value to the service via the named pipe opened in the first step.
- The service checks if the received value matches the value sent in the current session. If they match, the service opens the file containing the application hologram and sends it to the application via the named pipe.

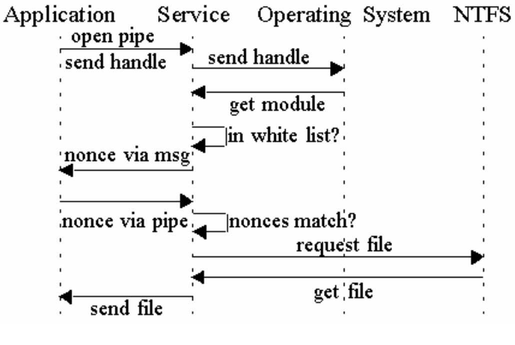


Figure 8: Communication diagram for retrieving an application hologram

At the end of the protocol run the application hologram has been transferred to the secure application that will display it on a confidentiality preserving secure surface. Hence, only trusted applications have access to the graphic, and the user can trust applications that show the graphic.

Using a named pipe ensures that the file is sent only to a benign application that opened the pipe. Sending a nonce via standard Windows messaging and associating it with the pipe session ensures that only an application present in the white list can request a hologram. Messages are put in the message queue of the respective application only.

As a precaution we use modified desktop ACLs as described earlier to disallow global hooks for other applications. The Win32 API otherwise offers to install a hook to retrieve all messages globally. Even then, an attack would have to be tailored to this specific protocol.

The same effect could also be achieved by using a file system filter driver identifying requests not only by user account but also by process.

4.5 Microsoft Windows Interface Changes

In the Microsoft Windows XP successor, code-named Longhorn, changes are anticipated as regards how applications use the desktop for displaying data. The Direct3D part of DirectX is expected to be used to render output. From a programmer's perspective, however, the new desktop composition engine will replace GDI and GDI+, but the interfaces will be similar. The desktop is still shared among applications. Hence, there may still be a need to acquire the display in exclusive mode to ensure a trusted output. We are not aware of changes to the input model.

Longhorn is currently expected to ship by 2005, so even if some problems with trusted output are solved with the new release, the need for practical solutions exists today. With the components we have presented here change can be applied to applications easily to heighten security.

NGSCB/Palladium (cf. England et al. 2003), Microsoft's attempt to add a separate secure kernel to Windows, also intends to provide a trusted path from an application to a user. However, this requires programming a new application since NGSCB works with a different application programming interface compared with today's Win32 API. In addition, NGSCB is not yet available.

5.0 RETROFITTING EXISTING APPLICATIONS

Today applications that would profit from a trusted path to the user exist. These applications can not be rewritten completely.

Edit and button controls in security sensitive dialog windows can be replaced by the DirectX-enhanced controls that we have presented in the preceding section. This replacement can take place at design/build time when a maintenance update or a new version of the application is produced. Since the controls are compatible with the standard controls, there is no need to change the source code depending on these input controls.

Replacing output requires some change to an application's source code. It may not be desired to switch to full-screen exclusive mode every time a message box is shown. Hence, only the security sensitive parts of the application have to be touched where displaying information to the user is important to be trusted.

If every message box can be replaced by our new component, i.e., if there are few, then another approach can be taken even at run time. A dynamic link library (DLL) could be injected into the process' address space. This DLL alters the address table for imported functions. Since our DXMessageBox function has the same signature as the Win32 API MessageBox function, all it takes is a change of the pointer to the function. However, changing function pointers from the outside without knowing the source code of the application may become a stability problem. We therefore favour minor changes in source code during build time.

We use version 8 of the DirectInput interface and version 7 of the DirectDraw interface. We have not investigated if it would be possible to revert to earlier versions of DirectX, namely versions 3 or 5. Hence we do not know whether our components would work with the earlier Windows NT 4 operating systems.

6.0 DISCUSSION OF THE SECURITY OF THE APPROACH

The security of our implementation of a trusted path for an application relies on the integrity of the operating system and resistance against system-wide global hooks.

DirectX is a part of the Windows 2000/XP operating system. As such, the operating system is responsible for the integrity of the DirectX modules. In addition, existing integrity protection tools could be used.

One of our implementation variants sends messages using the Windows messaging system. These messages are encrypted using AES, prohibiting other processes from inserting fabricated messages undetected in the stream.

To preclude processes from simulating input via the SendInput API, we offer multiple ways. One approach is to block use of the SendInput function by adding code to another process' memory and modifying the import address table. Simulated input can also be detected and dismissed by employing low level hooks that we use in our fast hook renewal method. Other processes have to be cut off from installing system-wide hooks. This is either done by fast hook renewal or by modifying the access control list of the desktop.

A hologram service is used to authenticate and authorize access of applications to a secret picture, called a hologram. Our protocol for communication with the service authenticates the application by a handle and a message. Again, system-wide hooks have to be denied other processes. An implementation variant could obviate a separate service, placing the functionality in a file system filter driver.

As stated earlier, our focus lies on protection against architectural vulnerabilities. If there are flaws owing to errors in the implementation of the platform, they have to be covered by other means.

Variants in the implementation, which we partly offer, help to increase resistance against generic attack tools. We offer three variants of our input components' implementation, and three variants as regards protection against simulated input, two of which can also be used for the hologram service.

7.0 CONCLUSION

Trojan horse programs, i.e., programs with additional hidden, often malicious, functions, are more and more popular forms of attack. Applications that execute in an insecure environment should have control over their communication with the user.

Our work solves integrity and authenticity of input, confidentiality, integrity and authenticity of output. Confidentiality of user input is not discussed and remains to be scrutinized.

We have in detail explored components for establishing a trusted path for an application in an event-driven system. These components can be adopted by developers to reinforce existing applications or to build new security sensitive applications.

We have presented a creative way of exploiting existing COTS components – DirectX – to directly access input and output devices. Compared with dedicated hardware or operating system replacements our solution gives tractable and cost-effective means to incorporate better protection into programs on desktop computers.

Different implementations of the components provide different levels of security and different points of attack, increasing resistance against generic attack tools. Current malicious software that threatens integrity of desktop user interaction often relies on weaknesses in the standard messaging system. These weaknesses are abated by our approach.

Further research should include metrics to measure the added security by employing COTS components and varying implementations. Probably additional standard user interface components like combo boxes or tool bars could be implemented using our approach to offer application developers more choices. It remains to be examined whether components could be built that also work with older versions of the Windows operating system family, i.e., NT4. It might also be of interest to explore if other application programming interfaces, e.g., the Qt framework or OpenGL, could be used in the same way as DirectX.

8.0 REFERENCES

[1] Balfanz, D. (2001). *Access Control for Ad-hoc Collaboration.* PhD thesis, Princeton University.

[2] Bergadano, F., Gunetti, D., and Picardi, C. (2002). 'User Authentication through Keystroke Dynamics'. ACM Transactions on Information and System Security 5.4(2002):367-397.

[3] Bråthen, R. (1998). 'Crash Course in X Windows Security'. *GridLock* 1(1998):1. http://www.hackphreak.org/gridlock/issues/issue.1/xwin.html

[4] Carlisle, M.C. and Studer, S.D. (2001). 'Reinforcing Dialog-Based Security'. *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security.* Pp. 24-29.

[5] CERT Coordination Center (1999). *CERT Advisory CA-99-02-Trojan-Horses*. http://www.cert.org/advisories/CA-1999-02.html

[6] Clark, D.D. and Wilson, D.R. (1987). 'A Comparison of Commercial and Military Computer Security Policies'. *Proceedings of 1987 IEEE Symposium on Security and Privacy.* Pp. 184-194.

[7] Cult of the Dead Cow (2003). *Back Orifice 2000.* http://bo2k.sourceforge.net

[8] Delphi-Jedi Project (2003). *DirectX headers and samples.* http://www.delphi-jedi.org

[9] Department of Defense (1985). *DoD 5200.28-STD Department of Defense Trusted Computer System Evaluation Criteria.* ('Orange Book')

[10] England, P., Lampson, B., Manferdelli, J., Peinado, M., and Willman, B. (2003). 'A Trusted Open Platform'. *Computer* 36.7(2003):55-62.

[11] Howard, M. (2002). *Tackling Two Obscure Security Issues.* http://msdn.microsoft.com/library/default.asp?url=/library/ en-us/dncode/html/secure08192002.asp

[12] Langweg, H. (2002). 'With Gaming Technology towards Secure User Interfaces'. *Proceedings of Annual Computer Security Applications Conference 2002.* Pp. 44-50.

[13] Microsoft (1998). *Microsoft Windows Architecture for Developers Training Kit.*

[14] Microsoft (2003). *Microsoft Developer Network Library.*

[15] Paget, C. (2002). 'Exploiting design flaws in the Win32 API for privilege escalation. Or... Shatter Attacks – How to break Windows'. http://www.google.com/search?q=cache:security.tombom.co.uk/shatter.html

[16] Rooker, T. (1993). 'Application Level Security Using an Object-Oriented Graphical User Interface'. *Proceedings of the 1992-1993 Workshop on New Security Paradigms.* Pp. 105-108.

[17] Schmid, M., Hill, F., and Ghosh, A.K. (2002). 'Protecting Data from Malicious Software'. *Proceedings of Annual Computer Security Applications Conference 2002.* Pp. 199-208.

[18] Spalka, A., Cremers, A.B., and Langweg, H. (2001). 'The Fairy Tale of »What You See Is What You Sign«. Trojan Horse Attacks on Software for Digital Signatures'. *Proceedings of IFIP Working Conference on Security and Control of IT in Society-II.* Pp. 75-86.

[19] Spalka, A., and Langweg, H. (2002). 'Notes on Program-Orientated Access Control'. *Proceedings of First International Workshop on Trust and Privacy in Digital Business – TrustBus.* Pp. 451-455.

[20] Thurrott, P. (2003). 'The Road To Windows 'Longhorn' Part Two'. Paul Thurrott's SuperSite for Windows. http://www.winsupersite.com/showcase/ longhorn_preview_2003.asp

[21] Tygar, J.D., and Whitten, A. (1996). 'WWW Electronic Commerce and Java Trojan Horses'. *Proceedings of the Second USENIX Workshop on Electronic Commerce.*

[22] Wiseman, S., Terry, P., Wood, A., and Harrold, C. (1988). 'The Trusted Path between SMITE and the User'. *Proceedings of 1988 IEEE Symposium on Security and Privacy.* Pp. 147-155.

[23] Xenitellis, S. (2002a). 'Security vulnerabilities in event-driven systems'. *Proceedings of IFIP SEC'2002.* Pp. 147-160

[24] Xenitellis, S. (2002b). 'A New Avenue of Attack: Event-driven System Vulnerabilities'. *Proceedings of European Conference on Information Warfare and Security, MCIL.* Pp. 177-185.

[25] Ye, E. and Smith, S. (2002). 'Trusted Paths for Browsers'. *Proceedings of 11th USENIX Security Symposium.* Pp. 263-279.

Building a Trusted Path for Applications Using COTS Components

Hanno Langweg
Norwegian Information Security Laboratory – NISlab
Department of Computer Science and Media Technology
Gjøvik University College

- Motivation for secure user interfaces
- Input
- Output
- Integration in existing applications

Motivation for secure user interfaces

Security sensitive applications, e.g. home banking, electronic signatures, online voting

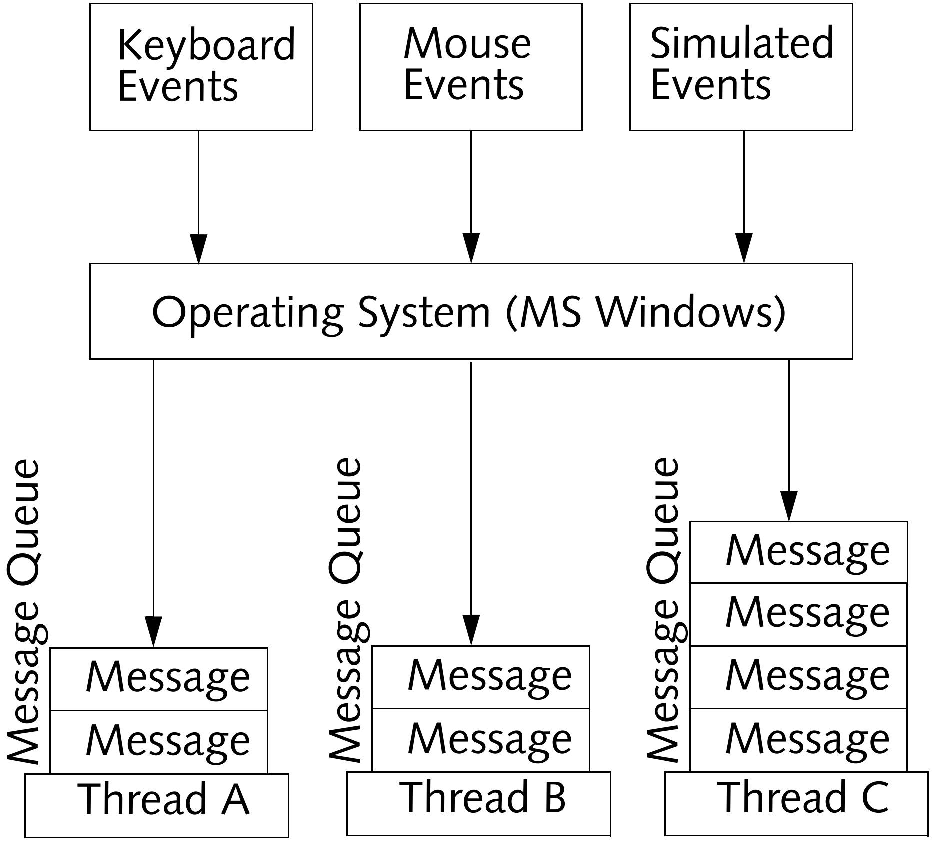
→ Need confirmation of user presence and user consent

Trusted path today

- Orange book (1985): “A mechanism by which a person at a terminal can communicate directly with the Trusted Computing Base (TCB). This mechanism can only be activated by the person or the TCB and can not be imitated by untrusted software.”
- User → Operating System: Ctrl+Alt+Delete to access Windows logon process
- Application → User: input can be simulated, output can be altered
- Distinguishing users and applications is required; not easily supported by current implementations of access control models
- Strict separation of applications often on expense of co-operation, ease of use

Input (i) – Traditional interface by MS Windows

Traditional input processing model in MS Windows



Message origin cannot be verified, simulation of events is possible (and convenient)

Input (ii) – Example “how to work around security“

Mailing list NTBugtraq 2004-03-12

- Anti-Spam tools' functionality broken by Office XP SP3 security update (iHateSpam/Sunbelt, Spamnet/Cloudmark)
- Confirmation dialog when third-party application uses Outlook
- Suggested work-around by spam tool vendor: Use Expresssoft's ClickYes program

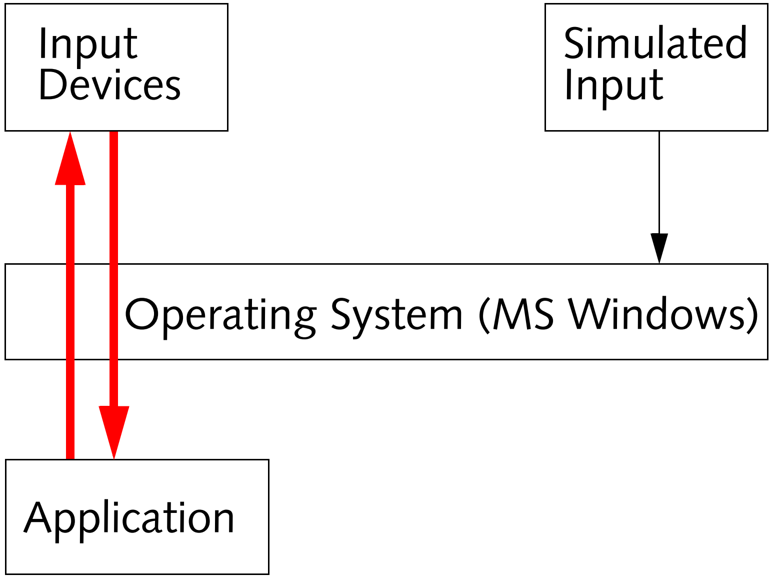
  “Express ClickYes 1.0.7 (updated on September 18, 2003)
  What is Express ClickYes

  Express ClickYes is a tiny program that sits in the taskbar and clicks the Yes button on behalf of you, when Outlook's Security Guard opens prompt dialog saying that a program is trying to send an email with Outlook or access its address book.”

  http://www.express-soft.com/mailmate/clickyes.html

Input (iii) – Interface for game developers

DirectX



- Intends to provide fast access to input devices, bypass messaging to react faster
- Polls devices directly, does not depend on Windows messages, fabricated messages do not have an effect
- Game developers like DirectX and employ it for user interfaces, hence it is a proven method

But...

- Provides integrity and authenticity, not confidentiality (no help against “key loggers”)
- SendInput API is still available to attackers

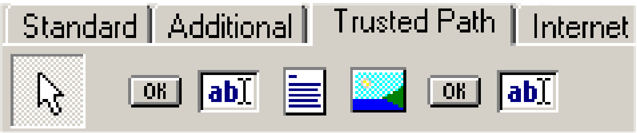
Input (iv) – Implementation of secure input

i) Determination of parts of the application that need more reliable input processing

- Not all situations require higher security
- Small set of security relevant operations, e.g.
  - ∗ Authorising an electronic signature
  - ∗ Entering a transaction code in a home banking application
- Source code changes incur costs

ii) Replacement of standard input processing

- DirectInput instead of/in addition to Windows messages
- Compatible components
  - ∗ Edit, Memo, Button
  - ∗ Different implementations possible

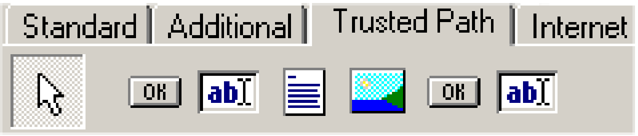


- Defending against SendInput: message hooks with fast renewal/modified desktop ACL, APP_Init dlls

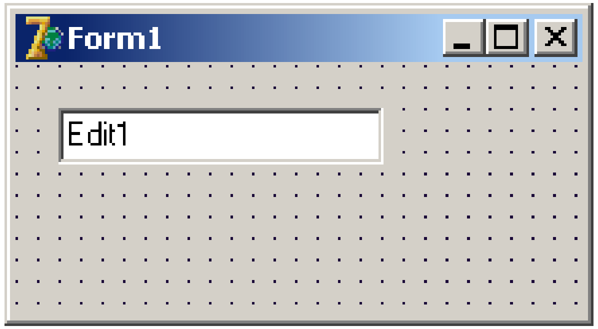
Input (v) – Example implementation

Replacing an edit control for text input:

- Use a sub-classed secure edit control from the component palette and build application like with an ordinary edit control



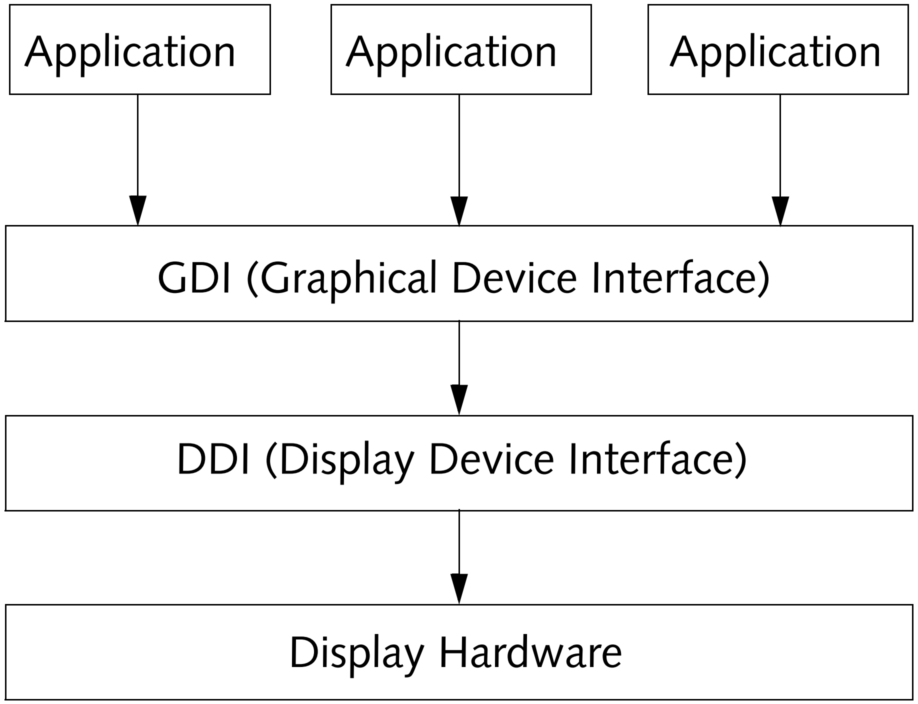
- User input works like before, may be a bit slower
  - ∗ Message confirmation (internal)
  - ∗ Message substitution (internal)
  - ∗ Input pre-processing by security service (external)



- Helper applications that fill the control with data will no longer work, neither will malicious programs
- Malicious programs can simulate (visible) user input using SendInput – that is more difficult; we need additional protection here: confining untrustworthy applications, modifying access rights, observing system usage
- Counter measures as operating system extensions or application only

Output (i) – Traditional interface by MS Windows

Traditional output processing model in MS Windows


Application
Application
Application
GDI (Graphical Device Interface)
DDI (Display Device Interface)
Display Hardware

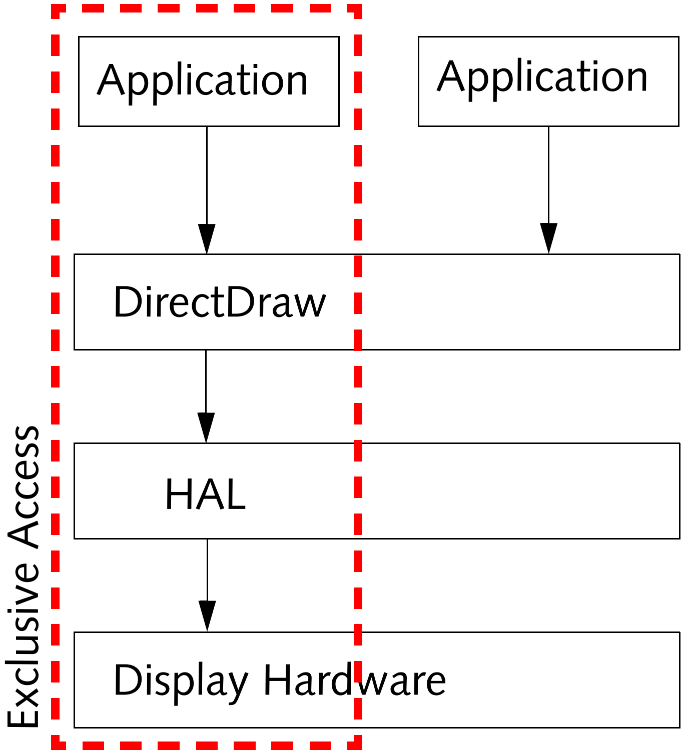
All applications on the same desktop are equal and can manipulate the display at their will

Assigning applications to separate desktops makes co-operation difficult (and expensive)

Output (ii) – Interface for game developers

DirectX

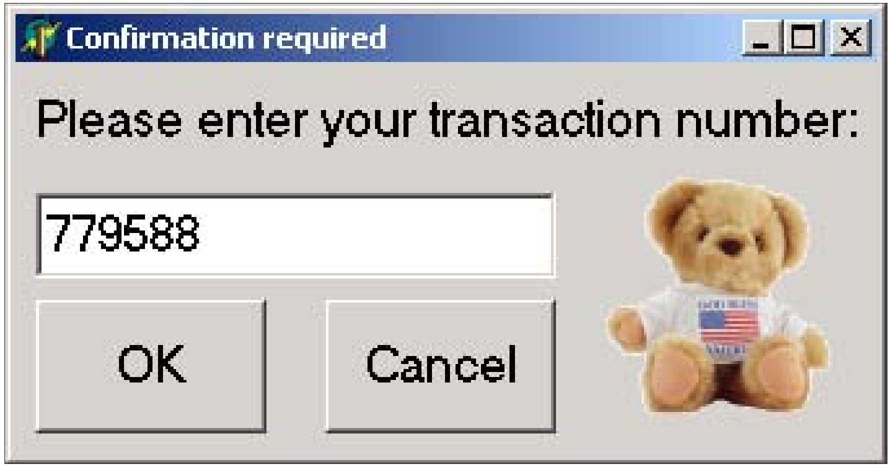
- Intends to provide fast access to output devices, more options for sophisticated graphical presentation
- Applications can request exclusive access to devices
  → exclusive access of output devices can be used to preserve integrity of output
- Game developers like DirectX and employ it for user interfaces, hence it is a proven method



Output (iii) – Implementation of secure output

i) Determination of parts of the application that need more reliable output processing

- Small set of security relevant operations, e.g.
  - ∗ Presenting data to be signed
  - ∗ Displaying secrets, e.g., transaction numbers for home banking
  - ∗ Proof of application authenticity
- Source code changes incur costs



Please enter your transaction number:

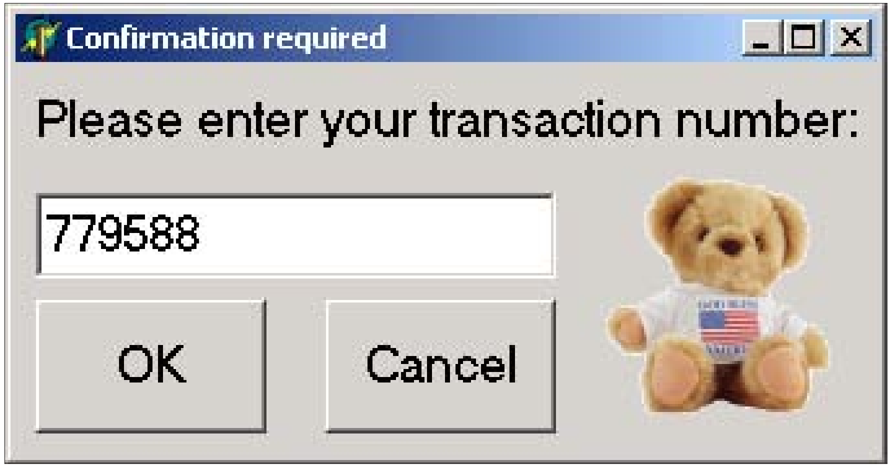
779588

ii) Replacement of standard output processing

- DirectDraw instead of GDI
- Compatible components
  - ∗ Window content, message box
- Yields integrity, confidentiality of output
- Monitoring of which application holds the focus

Output (iv) – Example of application authenticity

How to determine that you are using the correct application



Confirmation required

Please enter your transaction number:

779588

- Assign a secret (a picture) to an application that can only be shown by it and make sure that when shown it is not available to others: "application hologram"
- Presentation only on confidentiality-preserving surface
- Picture/file access only after verification of caller identity (process identifiable by window handle)
- Use service to add access rights on a per program basis on top of NTFS (could also be implemented as a file system filter driver)
- Have user verify the hologram and stop using the application when it is not present

Integration of secure input and output

i) Combining modified input and output methods

- Powerful user interfaces developed from scratch are possible using DirectX (cf. games)
- Replacing standard I/O components is possible:
  - ∗ Developed components for use with Borland Delphi as proof of concept
  - ∗ Restrictions in functionality compared with standard components (owes to implementation)
- Not difficult if used only when justified

ii) Integration in existing applications

- Determination of security sensitive parts of an application
- Replacing standard components via Drag&Drop in the development environment
- Modifying applications at run-time by pointer manipulation (technically possible but awkward to develop and debug; you do not want to do this)

Security of the approach

i) Protecting DirectX components against replacement/modification

- Components are part of the operating system
- Operating system is responsible for protection; additionally integrity preservation tools

ii) Protection against SendInput/simulated inputs

- Using low-level message hooks
  - ∗ Fast hook renewal
  - ∗ Modified desktop access control list
- APP_Init dynamic link libraries

iii) Protection against misuse by malicious programs

- Application hologram as proof of authenticity

Conclusions

COTS gaming technology can help:

- Authentic input: DirectInput
- Undisturbed output: DirectDraw

Advantages:

- Available on most machines, no additional hardware required
- Easy integration, low cost for development and deployment
- Co-operation between applications remains possible

Future work:

- Compatibility with older systems, i.e., NT4SP6a
- Different APIs, e.g., OpenGL, Qt, GTK
- Quantifying gain in security by employing COTS